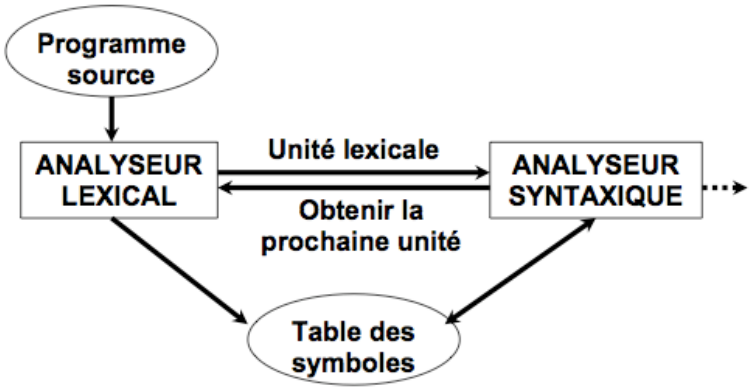
Chapitre 2

ANALYSE LEXICALE

1. INTRODUCTION

- La tâche principale d'un analyseur lexical est de lire les caractères d'entrée et de produire comme résultat une suite d'unités lexicales (*identificateurs, constantes réelles, entières, chaînes de caractères, opérateurs, séparateurs, mots clefs du langage,…) que l'analyseur syntaxique va utiliser.*



Interaction entre l'analyseur lexical et syntaxique

- À la réception d'une commande « prochaine unité lexicale », émanant de l'analyseur syntaxique, l'analyseur lexical lit les caractères d'entrée jusqu'à ce qu'il puisse identifier la prochaine unité lexicale.

- L'analyseur lexical réalise en plus certaines tâches secondaires comme :

  - L'élimination des caractères superflus (commentaires, espaces, passages à la ligne,…),

  - Identifier et traiter les parties du texte qui ne font pas partie à proprement parler du programme mais sont des directives pour le compilateur,

  - La gestion des numéros de lignes dans le programme source pour pouvoir associer à chaque erreur rencontré la ligne dans laquelle elle apparaît

2. UNITÉS LEXICALES, MODÈLES ET LEXÈMES

- Une **Unité Lexicale (UL)** est une suite de caractères qui a une signification collective.

Chaînes/ Mots / Lexèmes / Tockens	**Unités lexicales**
<,>, ≤, ≥,…	OPREL
Toto, ind, tab,...	IDENT
If, else, while,...	MOTCLE
;,.,(,),…	SEP

- Un **modèle** est une règle associée à une UL qui décrit l'ensembledes chaînes du programme qui peuvent correspondre à cette UL.

- On appelle **lexème** toute suite de caractères du programme source qui concorde avec le modèle d'une UL.

Unités lexicales	**Lexèmes**	**Description informelle des modèles**
IDENT	Truc, i, a1, ajouter_valeur, pi,…	Suite non vide de caractères composée de chiffres, lettres ou du symbole '_' et qui ne commencent pas par un chiffre
NOMBRE	-12, 83204, +0, …	Suite non vide de chiffres précédée éventuellement d'un seul caractère parmi {+, -}
REEL	12.4, 0.5E3, 10. -2.103E+2	Nombre suivi éventuellement d'un point et d'une suite (vide ou non) de chiffres. Le tout suivi éventuellement du caractère E ou e et d'un Nombre

3. ATTRIBUTS DES UNITÉS LEXICALES

- *OPREL correspond à la fois aux lexèmes : <, >, ≤, …*
  L'analyseur lexical doit fournir aux phases suivantes des informations additionnelles sur le lexème reconnu pour pouvoir distinguer entre les lexèmes d'une même unité lexicale.
- L'analyseur lexical réunit les informations sur les ULs dans des **attributs** (pointeur vers l'entrée de la table des symboles).

4. RECONNAISSANCE DES UNITÉS LEXICALES

- Pour décrire le modèle d'une UL, on utilisera des **expressions régulières / rationnelles.**
- Chaque modèle reconnaît un ensemble de **mots**.
- On compile une expression régulière en un reconnaisseur en construisant un automate à états finis déterministe ou non déterministe.

Algorithme: Simulation d'un AFD
Données : Une chaîne d'entrée x terminée par un caractère de fin de fichier fdf.
Un AFD D avec un état de départ e_0 et un { } d'états d'acceptation F.

Résultat : La réponse "oui" si D accepte x; "non" dans le cas contraire.
Méthode : La fonction transiter(e, c) donne l'état vers lequel il y a une transition depuis l'état e sur le caractère d'entrée c. La fonction CarSuiv retourne le prochain caractère de la chaîne d'entrée x.

```
e := e0
c:=CarSuiv();
Tanque (c ≠ fdf  & e≠null) Faire
        e:= Transiter (e, c);
        c:=  CarSuiv();
fin

 si (e ∈ F) alors
       retourner "oui"
sinon
       retourner "faux"
```

Exemple de reconnaisseur:

Morceau d'analyseur lexical pour le langage Pascal (en C) :

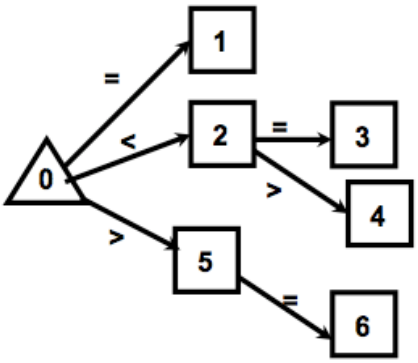
```
c=getchar();

Switch (c) {
                case'=' :   unite_lex=OPREL;
                            attribut=EGA;
                            break;

               case '<' :   unite_lex=OPREL; c=getchar();
                            if (c=='=')
                              {attribut=INFEG;}
                            else if (c=='>')
                                    {attribut=DIF;}
                                  else attribut=INF;
                            break;

               case …
```

ou encore en copiant le travail de l'automate (code moins rapide car il contient beaucoup d'appels de fonctions)




```
void etat0 ( )
{char c = getchar( ) ;
     Switch (c)
       {
        case'='              :etat1();
                               break;
        case'<'              :etat2();
                               break;
        case'>'              :etat5();
                               break;
        default: ERREUR();
        }
   }
`
```

```
Void etat2()
 {char c=getchar();
     Switch(c)
        {
         case‘=’              :etat3();
                                 break;
         case‘>’              :etat4();
                                 break;
         default: ERREUR();
         }
 }
```

Remarques

- Il n’est pas évident de s’y trouver si on veut modifier l’analyseur, le compléter,…

- Il existe des outils : (f)lex pour écrire des programmes simulant des automates à partir de simples DRs. Ci-dessous un exemple de programme (f)lex.

```
/* définitions des constants littérales */
%{
AFF, OPREL, MC_IF, NB, ID, INFEG
%}

Chiffre        [0-9]
Lettre         [a-zA-Z]
Entier         [+-] ? [0-9]{chiffre}*
Ident          {lettre}( {lettre}|{chiffre})*

%%
/* règles de traduction */
« := »                 {return AFF;}
« <= »                 {attribut = INFEG; return OPREL ;}
If | IF | if |iF       {return MC_IF;}
{entier}               {return NB ;}
{ident}                {return ID ;}
%%

/* bloc principal */
int yywrap() {return 1 ;}
main() {yylex() ;}
```